**Class:** Final Year (Computer Science and Engineering)

**Year:** 2025-26 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 4**

**Exam Seat No:**

**Title of practical:**

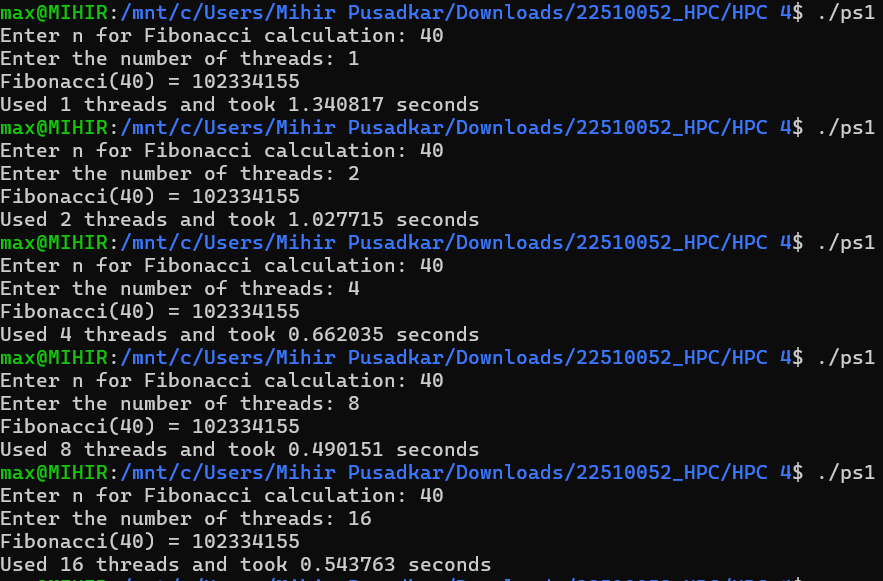
Study and Implementation of Synchronization

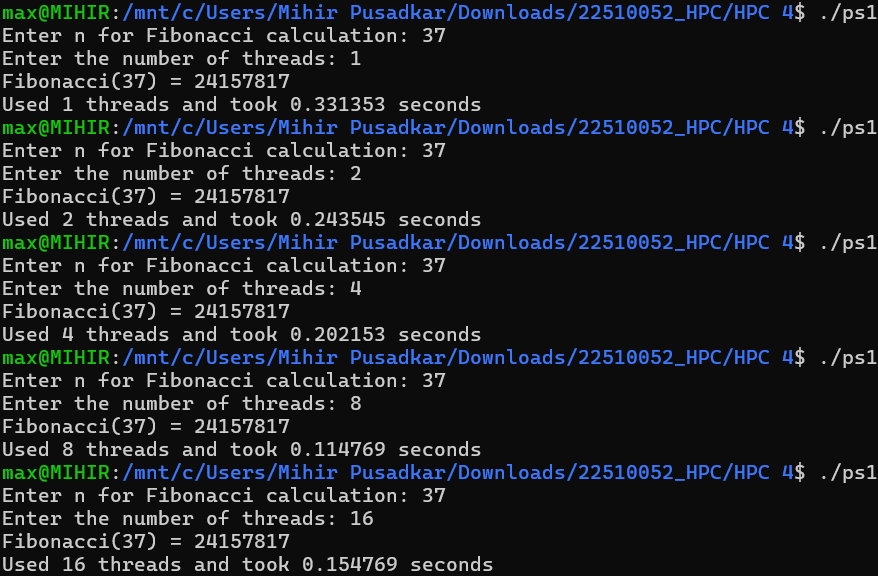
**Problem Statement 1:**

# Analyze and implement a Parallel code for below programs using OpenMP considering synchronization requirements. (Demonstrate the use of different clauses and constructs wherever applicable)

# Fibonacci Computation:

**Screenshots:**

****

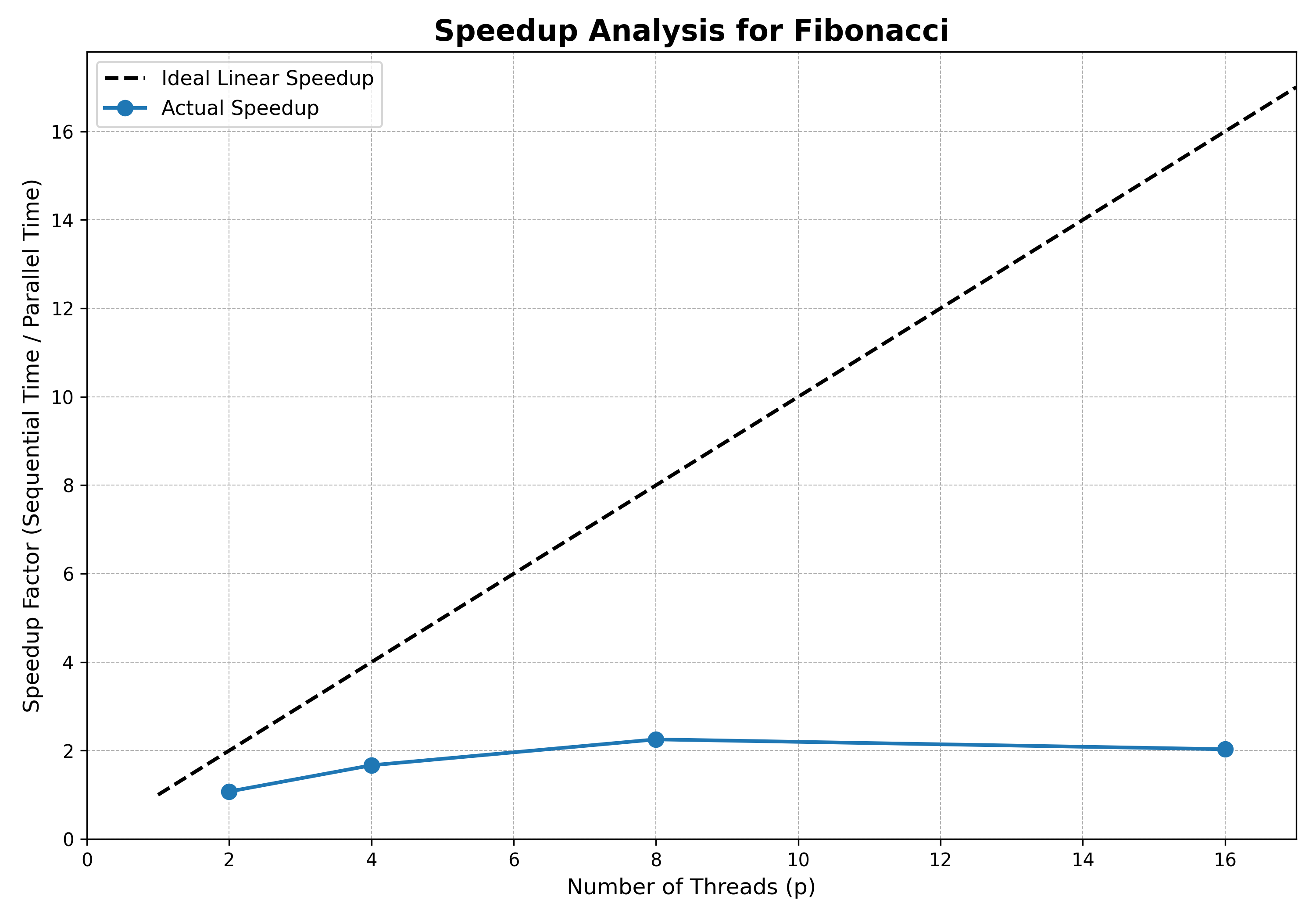
****

**Information:**

This program computes Fibonacci numbers using a parallel "divide and conquer" strategy based on the recursive formula fib(n) = fib(n-1) + fib(n-2). The implementation leverages OpenMP's task-based parallelism, where #pragma omp task is used to create independent tasks for the two recursive calls. A #pragma omp taskwait directive is used to synchronize and ensure both sub-problems are solved before their results are summed. This approach is computationally intensive due to redundant calculations, making it a strong candidate to demonstrate significant parallel speedup.

**Analysis:**

* **Overhead:** The cost of creating and managing parallel tasks remained high compared to the reduced parallel workload, limiting performance gains.
* **Performance:** The speedup flattened after 8 threads, indicating that adding more threads provided no further benefit and had reached the point of diminishing returns.

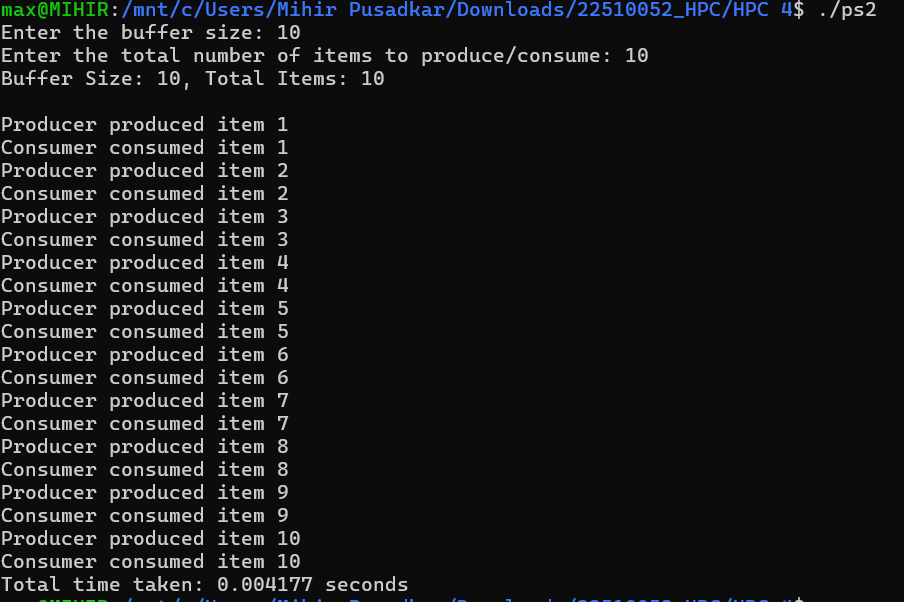
****

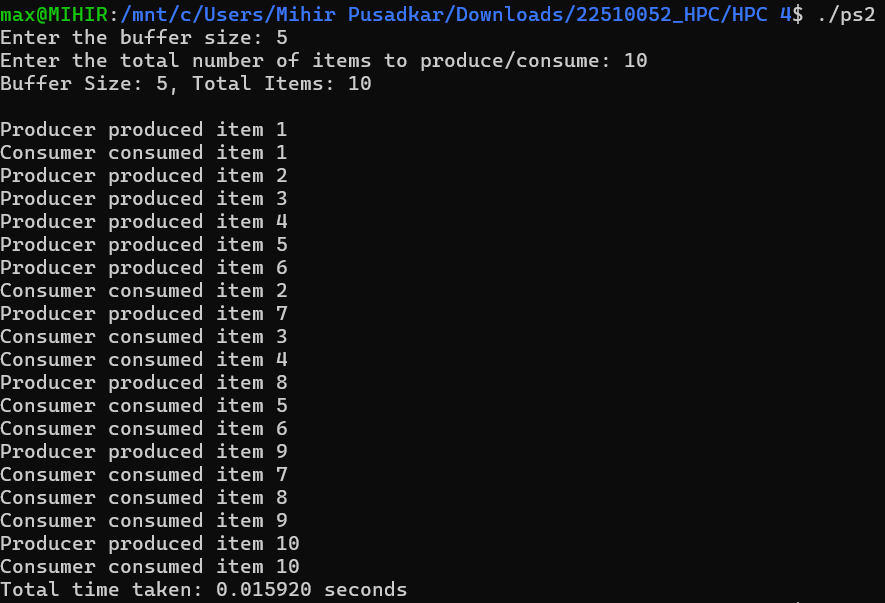
**Problem Statement 2:**

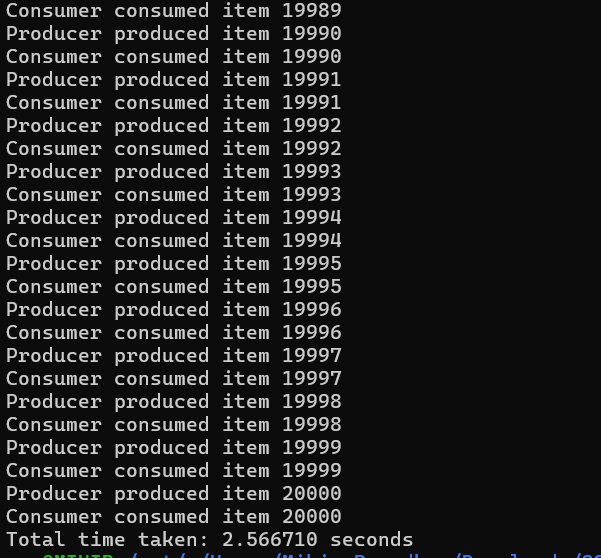
# Analyze and implement a Parallel code for below programs using OpenMP considering synchronization requirements. (Demonstrate the use of different clauses and constructs wherever applicable)

## Producer Consumer Problem

**Screenshots:**

****

****

****

**Information:**

This program implements a solution to Producer-Consumer problem. It uses one thread as a Producer to generate items and add them to a shared, fixed-size buffer. A second thread acts as a Consumer, removing items from the buffer. The implementation uses OpenMP constructs to manage the concurrent access to the shared buffer and to handle the boundary conditions where the buffer is either full or empty. Important constructs include #pragma omp parallel sections to run the two roles concurrently, #pragma omp critical to ensure mutually exclusive access to the buffer, and #pragma omp atomic for safe, efficient updates to the item counter.

**Analysis:**

The Producer-Consumer program demonstrates task parallelism, focusing on correct concurrent operation rather than speedup. The implementation successfully uses #pragma omp critical to prevent race conditions and handles buffer full/empty states, ensuring a safe data pipeline. Performance is measured by throughput, which is inherently limited by the necessary synchronization overhead, showcasing a classic concurrency pattern.

**Github Link:**